Propagation des contraintes tables souples Etude pr eliminaire

Christophe Lecoutre, Nicolas Paris, Olivier Roussel, Sébastien Tabary

Propagation des contraintes tables souples Etude préliminaire

Christophe Lecoutre **Nicolas Paris** **Olivier Roussel** **Sébastien Tabary**

CRIL - CNRS UMR 8188,
Université Lille Nord de France, Artois,
rue de l'universite, 62307 Lens cedex, France
{lecoutre,paris,roussel,tabary}@cril.fr

Résumé

Durant ces dix dernières années, de nombreuses études ont étés réalisées pour le cadre WCSP (Weighted Constraint Satisfaction Problem). En particulier, ont été proposées de nombreuses techniques de filtrage basées sur le concept de cohérence locale souple telle que la cohérence de nœud, et surtout la cohérence d'arc souple. Toutefois, la plupart de ces algorithmes ont étés introduits pour le cas des contraintes binaires, et la plupart des expérimentations ont étés menées sur des réseaux de contraintes comportant uniquement des contraintes binaires et/ou ternaires. Dans cet article, nous nous intéressons aux contraintes tables souples de grande arité. Nous proposons un premier algorithme pour filtrer ces contraintes et nous l'intégrons à PFC-MRDAC. Bien que nous soyons encore dans la phase préliminaire de nos travaux, nous montrons que cet algorithme peut être utile pour résoudre efficacement un nouveau jeu d'instances de mots-croisés.

Abstract

WCSP is a framework that has attracted a lot of attention during the last decade. In particular, there have been many developments of filtering approaches based on the concept of soft local consistencies such as node consistency (NC), arc consistency (AC), full directional arc consistency (FDAC), existential directional arc consistency (EDAC), virtual arc consistency (VAC) and optimal soft arc consistency (OSAC). Almost all algorithms related to these properties have been introduced for binary weighted constraint networks, and most of the conducted experiments typically include constraint networks involving only binary and ternary constraints. In this paper, we focus on extensional soft constraints of large arity. We propose an algorithm to filter such constraints and embed it in PFC-MRDAC. Despite still being preliminary at this stage, we show it can be used to solve efficiently a new benchmark of Crossword puzzles.

1 Introduction

Le problème de satisfaction de contraintes pondéré (WCSP) consiste, pour un réseau constitué de variables et de contraintes souples, à trouver une affectation de valeur à chaque variable qui soit optimale, c'est à dire une assignation complète de coût minimal parmi toutes les assignations complètes possibles. Il s'agit d'un problème NP-difficile. Un certain nombre de travaux ont étés menés pour adapter au cadre WCSP des propriétés (et algorithmes efficaces) définies pour le cadre CSP, généralement dans le but de filtrer l'espace de recherche, comme par exemple la cohérence de nœud (NC) ou la cohérence d'arc (AC) [11, 14]. Un certain nombre d'algorithmes de plus en plus sophistiqués ont été proposés au cours des années pour approcher la cohérence d'arc souple idéale : FDAC, EDAC, VAC et OSAC (voir [6]).

Les algorithmes évoqués ci-dessus utilisent des opérations de transfert de coûts, ce qui les rend particulièrement efficaces pour résoudre des instances de problèmes binaires et/ou ternaires. Pour des contraintes de plus grande arité, un problème d'ordre combinatoire se présente. Une première solution à ce problème est de retarder la propagation (des coûts) en attendant qu'un nombre suffisant de variables soient affectées, mais cela réduit considérablement la capacité de filtrage des algorithmes en début de recherche. Une seconde solution est de concevoir des adaptations des algorithmes de cohérence d'arc souple pour certaines familles de contraintes (globales) souples. C'est l'approche proposée dans [17] où le concept de contraintes

saines pour la projection est introduit. Enfin, une troisième solution [9] est la décomposition de fonctions de coûts en fonction de coûts d'arité plus faible. Toutefois, toutes les fonctions de coût ne sont pas décomposables. Dans cet article, nous proposons une première approche pour les contraintes tables souples (i.e. les contraintes souples définies en extension) en exploitant le principe de la réduction tabulaire simple (STR) [19]. À ce stade, notre approche ne permet pas les transferts de coût et se trouve donc intégrée à l'algorithme classique PFC-MRDAC [12].

2 Préliminaires

Un *réseau de contraintes* (CN) P est un couple $(\mathscr{X}, \mathscr{C})$ où $\mathscr{X}$ est un ensemble fini de n variables et $\mathscr{C}$ est un ensemble fini de e contraintes. Chaque variable x a un domaine associé noté $dom(x)$, qui contient l'ensemble fini des valeurs pouvant être assignées à x ; le domaine initial de x est noté $dom^{init}(x)$. d représente la taille du plus grand domaine. Chaque contrainte c_S porte sur un ensemble ordonné $S \subseteq \mathscr{X}$ de variables appelé *portée* de c_S, et définie par une relation contenant l'ensemble des tuples autorisés pour les variables de S. L'arité d'une contrainte est le nombre de variables dans sa portée. Une contrainte *unaire* (resp., *binaire*) porte sur 1 (resp., 2) variable(s), et une contrainte *non-binaire* porte sur plus de deux variables. Une *instanciation* I d'un ensemble $X = \{x_1, \ldots, x_p\}$ de variables est un ensemble $\{(x_1, a_1), \ldots, (x_p, a_p)\}$ tel que $\forall i \in 1..p, a_i \in dom^{init}(x_i)$; chaque a_i est noté $I[x_i]$. I est *valide* sur P ssi $\forall (x, a) \in I, a \in dom(x)$. Une solution de P est une instanciation complète de P (i.e., l'assignation d'une valeur à chaque variable) qui satisfait toutes les contraintes. Pour plus d'information sur les réseaux de contraintes, voir [8, 15].

Un *réseau de contraintes pondéré* (WCN) P est un triplet $(\mathscr{X}, \mathscr{C}, k)$ où $\mathscr{X}$ est un ensemble fini de n variables, comme pour un CN, $\mathscr{C}$ est un ensemble fini de e contraintes souples, et k est soit un entier naturel strictement positif soit $+\infty$. Chaque contrainte souple $c_S \in \mathscr{C}$ porte sur un ensemble ordonné S de variables (sa portée) et est définie par une fonction de coût de $l(S)$ vers $\{0, \ldots, k\}$, où $l(S)$ est le produit cartésien des domaines des variables présentes dans S ; pour toute instanciation $I \in l(S)$, on notera le coût de I dans c_S par $c_S(I)$. Pour simplifier, pour toute contrainte (souple) c_S, un couple (x, a) avec $x \in S$ et $a \in dom(x)$ est appelé une *valeur* de c_S. Une instanciation de coût k (noté aussi $\top$) est interdite. Autrement, elle est autorisée avec le coût correspondant (0, noté aussi $\bot$, est complètement satisfaisant). Les coûts sont combinés par l'opérateur binaire $\oplus$ défini par :

$$\forall a, b \in \{0, \ldots, k\}, a \oplus b = min(k, a + b)$$

L'objectif du problème de satisfaction de contraintes pondéré (WCSP) est, pour un WCN donné, de trouver une instanciation complète de coût minimal. Pour plus d'information sur les contraintes pondérées (valuées), voir [2, 18].

Différentes variantes de la cohérence d'arc souple pour le cadre WCSP ont étés proposées durant ces dix dernières années. Il s'agit de AC* [11, 14], la cohérence d'arc directionnelle complète (FDAC) [4], la cohérence d'arc directionnelle existentielle (EDAC) [7], la cohérence d'arc virtuelle (VAC) [5] et la cohérence d'arc souple optimale (OSAC) [6]. Tous les algorithmes proposés pour atteindre ces différents niveaux de cohérence utilisent des opérations de transfert de coûts (ou transformations préservant l'équivalence) telles que la projection unaire, la projection et l'extension. Une autre approche classique consiste à ne pas effectuer de transferts de coûts. Il s'agit de l'algorithme PFC-MRDAC [10, 13, 12] qui est un algorithme de séparation et évaluation (branch and bound) permettant de calculer des bornes inférieures à chaque nœud de l'arbre de recherche. À chaque nœud, on calcule un minorant (lb), sous estimation du coût de n'importe quelle solution pouvant être atteinte à partir du nœud courant. On considère une partition [1] *part* de l'ensemble des contraintes qui à chaque variable x associe un élément noté $part(x)$ de cette partition (i.e., chaque contrainte est associée à exactement une variable). Pour une valeur (x, a), il est possible de calculer un minorant $lb(x, a)$ comme suit : $lb(x, a) =$

$$distance + \tag{1}$$

$$\sum_{c_S \in part^{nc}(x)} minCosts(c_S, x, a) + \tag{2}$$

$$\sum_{y \neq x} \min_{b \in dom(y)} \sum_{c'_S \in part^{nc}(y)} minCosts(c'_S, y, b) \tag{3}$$

où :

- *distance* représente le coût des contraintes couvertes, i.e. la somme du coût des contraintes dont la portée ne comporte que des variables instanciées (explicitement par l'algorithme de recherche arborescente) ;
- $part^{nc}(x)$ représente $part(x)$ sans les contraintes couvertes (pour ne pas les compter plus d'une fois) ;
- $minCosts(c_S, x, a)$ représente le coût minimal dans c_S de toute instanciation valide comportant (x, a) ; plus formellement $minCosts(c_S, x, a) = min\{c_S(I) \mid I \in l(S) \wedge I[x] = a\}$.

1. On suppose que chaque variable est impliquée dans au moins une contrainte.

La valeur de $lb(x, a)$ est calculée en prenant en compte le coût des contraintes couvertes (équation 1), celui des contraintes (non couvertes) associées à x (équation 2), et celui des contraintes (non couvertes) associées à d'autres variables que x (équation 3). Si $lb(x, a)$ est supérieur ou égal à la borne supérieure (ub) qui représente le coût de la meilleure solution trouvée, alors il est possible de supprimer (x, a).

3 Algorithme

Dans le cadre CSP, une contrainte table (positive) c_S est une contrainte définie en extension par la liste de ses tuples autorisés ; nous notons celle-ci $table[c_S]$ et utilisons comme indices $1..t$ où t désigne le nombre de tuples. L'un des algorithmes de filtrage les plus efficaces pour les contraintes table (pour le cadre CSP) est appelé STR [19, 16]. STR permet de maintenir en permanence la liste des supports de chaque contrainte table ; cette liste est appelée la table courante. Pour une contrainte table donnée, cette liste est gérée à l'aide des structures suivantes :

- $position[c_S]$ est un tableau de taille t qui fournit un accès indirect aux tuples de $table[c_S]$. À tout moment, les valeurs dans $position[c_S]$ représentent une permutation de $\{1, 2, \ldots, t\}$. Le $i^{\text{ème}}$ tuple de la contrainte est $table[c_S][position[c_S][i]]$.
- $currentLimit[c_S]$ est la position du dernier tuple (support) courant dans $table[c_S]$. La table courante de c_S contient exactement $currentLimit[c_S]$ tuples. Les valeurs dans $position[c_S]$ aux indices allant de 1 à $currentLimit[c_S]$ sont les positions des tuples courants de c_S.

Une contrainte table souple c_S est une contrainte définie par une liste $table[c_S]$ de tuples accompagnée d'une liste $costs[c_S]$ de leur coût, ainsi que d'un coût par défaut $default[c_S]$ pour tous les tuples non représentés explicitement. L'algorithme PFC-MRDAC nécessite de calculer pour toute contrainte c_S et toute valeur (x, a) de c_S le coût minimal de tout tuple valide τ tel que $\tau[x] = a$. Nous montrons comment effectuer ce calcul en adaptant STR aux contraintes tables souples grâce aux structures suivantes :

- $minCosts[c_S]$ est un tableau (à 2 dimensions) qui donne le coût minimal de tout tuple pour chaque valeur (x, a) de c_S.
- $nbTuples[c_S]$ est un tableau (à 2 dimensions) qui donne le nombre de tuples valides (parmi ceux représentés explicitement dans la table initiale) pour chaque valeur (x, a) de c_S.

WSTR, i.e. STR pour le cadre WCSP, est décrit par l'algorithme 1. Cet algorithme est appelé chaque fois qu'un évènement nécessite de mettre à jour la table courante et donc les coûts minimaux de chaque valeur. Tout d'abord, les structures $minCosts[c_S]$ et $nbTuples[c_S]$ sont initialisées aux lignes 1 à 4. Ensuite, chaque tuple τ (ligne 8) de la table courante et son coût γ (ligne 9) sont considérés. Si le tuple est toujours valide (ligne 10, appel à l'algorithme 2) alors on met à jour les structures $minCosts[c_S]$ et $nbTuples[c_S]$ (lignes 11 à 15). Sinon, on élimine le tuple en échangeant la position des tuples aux positions i et $currentLimit[c_S]$ (ligne 18) et en décrémentant ensuite $currentLimit[c_S]$ (ligne 19). Pour finir, il faut prendre en compte les tuples non représentés explicitement (lignes 20 à 25) : si le nombre de tuples valides trouvé pour une valeur

Algorithm 1: WSTR(c_S : contrainte souple)

```
1  foreach variable x ∈ S do
2      foreach a ∈ dom(x) do
3          minCosts[c_S][x][a] ← ub
4          nbTuples[c_S][x][a] ← 0
5  i ← 1
6  while i ≤ currentLimit[c_S] do
7      index ← position[c_S][i]
8      τ ← table[c_S][index]
9      γ ← costs[c_S][index]
10     if isValidTuple(c_S, τ) then
11         foreach variable x ∈ S do
12             a ← τ[x]
13             nbTuples[c_S][x][a] + +
14             if γ < minCosts[c_S][x][a] then
15                 minCosts[c_S][x][a] ← γ
16         i ← i + 1
17     else
18         swap(position[c_S], i, currentLimit[c_S])
19         currentLimit[c_S] − −
20 foreach variable x ∈ S do
21     nb ← |Π_{y∈scp(c)|y≠x} dom(y)|
22     foreach a ∈ dom(x) do
23         if nbTuples[c_S][x][a] ≠ nb then
24             if default[c_S] < minCosts[c_S][x][a]
               then
25                 minCosts[c_S][x][a] ← default[c_S]
```

Algorithm 2: isValidTuple(c_S, τ) : booléen

```
1  foreach variable x ∈ S do
2      if τ[x] ∉ dom(x) then
3          return false
4  return true
```

(x, a) dans la table courante est différent du nombre de tuples valides possibles (calculé à la ligne 21) pour cette valeur alors il existe au moins un tuple pour (x, a) avec ce coût et donc le coût par défaut doit être considéré.

La complexité spatiale de WSTR pour une contrainte table souple c_S, avec r désignant l'arité de c_S et t désignant le nombre de tuples explicites de la table (initiale) de c_S, est donnée par la complexité spatiale des structures de données propres à STR, à savoir $O(n + rt)$ [16], et la complexité spatiale des structures additionnelles $minCosts[c_S]$ et $nbTuples[c_S]$ qui est $O(rd)$. On obtient donc globalement $O(e(n + r.max(d, t)))$. La complexité temporelle de l'algorithme 1 est $O(rd)$ pour les lignes 1-4, $O(tr)$ pour les lignes 5-19, et $O(rd)$ pour les lignes 20-25, soit globalement $O(rd + tr)$.

Un certain nombre de précisions sont à apporter sur cet algorithme. Tout d'abord, dans cet article, pour simplifier il est présenté dans le contexte d'une utilisation hors recherche. Ensuite, il est certainement possible d'apporter un certain nombre d'améliorations selon la distribution des coûts (en particulier, lorsque $defaut[c_S] = 0$ ou $defaut[c_S] = k$). Pour finir, l'intérêt de cet algorithme réside dans son efficacité lorsqu'une réduction importante de la table est constatée, et la possibilité de restaurer les tuples en temps constant.

4 Résultats expérimentaux

Bien sûr, nous avons cherché à valider expérimentalement cette première approche que nous proposons pour filtrer les contraintes tables souples. Malheureusement, à notre connaissance aucun jeu d'essai comportant des instances WCSP avec des contraintes tables d'arité importante n'est disponible. Nous avons donc développé une nouvelle série d'instances de mots-croisés, appelée *crossoft*, qui se prêtent naturellement à une expression "souple". Pour cela, nous avons fusionné deux dictionnaires (appelés OGD), l'un comportant des noms communs et l'autre des noms propres. Le coût d'un mot (tuple) a été calculé de la manière suivante :

- mot commun : coût 0
- mot propre : coût r où r est la longueur du mot

À l'aide de ce nouveau dictionnaire (souple), l'objectif est alors de remplir des grilles de mots-croisés vides en minimisant le coût global. Le type de pénalités considérées ici ne correspond pas exactement aux bénéfices associés aux mots tels que décrits sur le site `http://ledefi.pagesperso-orange.fr` mais il s'en rapproche (et a le mérite de la simplicité). Nous avons généré des instances pour les séries de grilles appelées *herald*, *puzzle*, et *vg*.

Notre premier objectif ici est de montrer que l'approche WSTR peut s'avérer efficace pour résoudre des instances assez difficiles (au regard de la difficulté des instances dures). Nous avons aussi voulu nous assurer que retarder la propagation des algorithmes de filtrage AC* et EDAC (c'est-à-dire attendre qu'il reste au plus 3 ou 4 variables non assignées avant de les exécuter) n'était pas une solution viable (ne pas retarder la propagation l'est encore moins par rapport aux instances que nous avons testées, étant données l'arité des contraintes et la taille des domaines des variables). C'est pourquoi, pour montrer le potentiel de WSTR, nous avons comparé les performances des algorithmes suivants :

- WSTR, l'algorithme 1 utilisé au sein de PFC-MRDAC
- maintenir AC*, avec propagation retardée
- maintenir EDAC, avec propagation retardée

L'heuristique de choix de variable que nous avons choisie est *dom/ddeg* [1] (plus stable que *dom/wdeg* [3] dans un but de comparaison), et l'heuristique de choix de valeur consiste à toujours prendre une valeur de coût minimum. Les expérimentations ont été menées grâce à notre solveur AbsCon sur un cluster équipé de processeurs Intel(R) Xeon(TM) CPU 3.00GHz. Le temps limite alloué pour résoudre chaque instance est de 1200 secondes. Résoudre une instance signifie ici trouver une solution optimale et prouver qu'il s'agit de l'optimum.

Le tableau 1 fournit quelques résultats concernant la résolution des instances *crossoft*. Ces instances comportent toutes des contraintes d'arité supérieure ou égale à 5. Pour chaque instance (à noter que r_{max} désigne l'arité maximale), le temps CPU (cpu) total pour la résoudre est donné ainsi que le nombre de nœuds explorés (nœuds). Les résultats obtenus montrent bien que WSTR est plus efficace que maintenir AC* et EDAC (avec propagation retardée) sur des problèmes de grande arité. Il ne faut toutefois pas être surpris par ces résultats, puisque nous savons que AC* et EDAC (sous leurs formes génériques) ne sont pas adaptés à ces types de problèmes.

5 Conclusion

Dans ce papier, nous avons montré que l'algorithme de filtrage WSTR, intégré à l'algorithme classique PFC-MRDAC, est adapté aux instances WCSP comportant des contraintes tables souples de grande arité. Sans doute, de nombreuses améliorations sont possibles, incluant la possibilité de spécialiser l'algorithme selon la distribution des coûts, et aussi la possibilité sous certaines hypothèses d'utiliser les opérations de transferts de coûts.

Instances	n	e	d	r_{max}		AC*	EDAC	WSTR
ogd-05-03	21	10	26	5	cpu	$1,77$	$1,94$	$0,8$
					nœuds	46	46	182
ogd-05-08	21	10	26	5	cpu	> 1200	$2,81$	$0,75$
					nœuds		317	102
ogd-05-09	19	10	26	5	cpu	$0,98$	> 1200	$0,74$
					nœuds	37		63
ogd-puzzle-04	19	10	26	5	cpu	$1,2$	$1,47$	$0,82$
					nœuds	19	19	54
ogd-puzzle-05	21	10	26	5	cpu	> 1200	> 1200	$0,8$
					nœuds			88
ogd-puzzle-11	38	18	26	7	cpu	$20,7$	> 1200	$1,75$
					nœuds	$271K$		553
ogd-vg-4-7	28	11	26	7	cpu	$14,0$	$14,1$	$1,17$
					nœuds	7351	7351	222
ogd-vg-4-8	32	12	26	8	cpu	645	> 1200	$1,48$
					nœuds	$1398K$		513
ogd-vg-5-5	25	10	26	5	cpu	> 1200	> 1200	$0,9$
					nœuds			232

TABLE 1 – Temps CPU et nombre de nœuds visités pour prouver l'optimalité sur quelques instances *crossoft*.

Remerciements

Ce travail bénéficie du soutien du CNRS et d'OSEO dans le cadre du projet ISI Pajero.

Références

[1] C. Bessiere and J. Régin. MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.

[2] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based CSPs and valued CSPs : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999.

[3] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.

[4] M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3) :311–342, 2003.

[5] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258, 2008.

[6] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8) :449–478, 2010.

[7] S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa. Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. In *Proceedings of IJCAI'05*, pages 84–89, 2005.

[8] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[9] A. Favier, S. de Givry, A. Legarra, and T. Schiex. Pairwise decomposition for combinatorial optimization in graphical models. In *Proceedings of IJCAI'11*, pages 2126–2132, 2011.

[10] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3) :21–70, 1992.

[11] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[12] J. Larrosa and P. Meseguer. Partition-Based lower bound for Max-CSP. *Constraints*, 7 :407–419, 2002.

[13] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1) :149–163, 1999.

[14] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

[15] C. Lecoutre. *Constraint networks : techniques and algorithms*. ISTE/Wiley, 2009.

[16] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraint. *Constraints*, 16(4) :341–371, 2011.

[17] J. Lee and K. Leung. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In *Proceedings of IJCAI'09*, pages 559–565, 2009.

[18] P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In *Handbook of Constraint Programming*, chapter 9, pages 281–328. Elsevier, 2006.

[19] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177 :3639–3678, 2007.